Reducing the Physical Cost of Large Register Files in EPIC Architectures with Stacked Register Aliasing

Ron Arnold
Hewlett Packard Co.
rarnold@hp.com

Rohit Bhatia
Hewlett Packard Co.
rohit.bhatia@hp.com

Don Soltis
Hewlett Packard Co.
don.soltis@hp.com

Abstract

Large register files are a key feature of EPIC architectures. Such large register files coupled with wide instruction issue, also typical of EPIC machines, present an interesting design challenge. This paper presents a register aliasing technique to reduce the physical cost in terms of area, power and cycle time at the expense of a small performance loss. The register aliasing technique described can be applied to the hazard detection logic of a processor to reduce the complexity of this logic. Performance simulation results presented show a negligible loss for SpecINT95 suite and moderate loss for SpecFP95 suite. It is also shown how the physical register file size can actually be increased through the use of stacked register aliasing.

1. Introduction

EPIC architectures, such as the Itanium® architecture, provide key features to enhance instruction level parallelism and support explicit parallelism. Explicit parallelism is supported by large parallel execution resources and large register files. Speculation and predication support allow compilers to enhance ILP. Predication allows conditional execution without branches implying larger basic blocks. Both of these ILP enhancement features tend to require a larger set of registers. The Itanium® architecture provides a 128-entry integer and a 128-entry floating-point register file. Most conventional RISC architectures feature only 32-entry register files.

Although large register files are key ingredients of EPIC performance delivery, they present an interesting challenge for EPIC hardware designers in the face of ever shrinking processor cycle time. The design challenge is presented both in terms of register file access time and register hazard detection for pipelined processors. In addition, there is a need to support wide instruction issue.

The large number of physical registers in Itanium® architecture compounds an already difficult problem in the physical implementation, namely hazard detection. With increasing numbers of execution pipelines and increasing pipeline lengths, the number of in-flight data dependencies can be daunting. The complexity of detecting these myriad data hazards has begun to require non-traditional hardware methods.

In the register read (REG) pipeline stage all read-after-write (RAW) and write-after-write (WAW) data hazards must be detected in order resolve data dependency hazards. The processor must stall instructions whose source data will not be available as they enter the execution pipeline stage, as well as instructions whose results might otherwise be overwritten by older, longer latency, instructions. This is performed by comparing source and destination register identifiers (regids) in REG verses the destination regids of all data producers in later pipeline stages, up to the stage where the register file is actually written. These compares are not required for all producer-producer and producer-consumer combinations; the number is greatly reduced by data bypassing.

In the Itanium® 2 processor, while considerable bypassing is provided between stages and pipelines, there are still 720 regid compares required to detect the integer, or general register (GR) and floating-point register (FR) data hazards which can stall the instructions in the REG stage [3]. Each compare has up to 6 qualifiers, such as predicate values, instruction valids, and instruction types, which must be evaluated in order to determine if a register ID match constitutes a hazard. The magnitude of the task precludes single-cycle hazard detection using explicit numerical comparators and gating logic.

The producers and consumers in the REG stage must also be evaluated verses long-latency loads which have committed or retired but whose data is still unavailable. This is often performed using a load scoreboard, effectively an array of 1-bit registers with multiple read and write ports. There is one entry for each register in a given register file, which, when asserted, indicates a load

is pending a write to its corresponding register. Each consumer and producer must be provided a read port to determine pending writes to their targets. Each memory load pipeline must have a write port to set scoreboard entries as a load retires, and also a write port to clear scoreboard entries as data returned from the memory subsystem is finally written to the register file. The Itanium® 2 microarchitecture would require a 127x1 register file with 18 read ports and 6 write ports for the GRs alone. A similar scoreboard structure would be required for FR scoreboarding.

The Itanium® architecture provides 128 integer registers. Registers r0-r31 are always accessible and are known as static registers. The static registers are similar to the 32 general registers in most RISC architectures such as PA-RISC and PowerPC. The remaining 96 registers, r32-r127, are called stacked registers. The Itanium® architecture allows for more than 96 physical registers to be used to implement the stacked registers; however, only a maximum of 96 are visible architecturally at any given time [6].

This paper presents an aliasing technique, which is applied to the stacked registers only in an attempt to simplify the pipeline hazard detection logic with minimal performance cost. The physical complexity of the data hazard detection logic or the scoreboard is governed by the number of pipeline stages and the number of registers that need to be tracked. By aliasing the stacked register set, one can reduce the complexity and size of the hazard detection logic. This will lead to occasional false hazard detection and consequently false pipeline stalls but does not impact functional correctness. Furthermore, by utilizing stacked register aliasing, hardware implementations can choose to grow the number of physical registers corresponding to the stacked registers with no increase in the hazard detection logic. The increased physical registers can offset the performance loss due to false pipeline stalls by reducing the spills and fills of stacked registers.

This paper is organized as follows. The next section provides background for the register stack engine and rotating and stacked register concepts in Itanium® architecture. We also provide physical design background for hazard detection logic in Itanium® processor implementations. The third section details the concepts of stacked register aliasing and shows how the complexity of the hazard detection logic is reduced. Then, we present results of our performance simulations showing the small performance loss for integer programs.

2. Background

The Itanium® architecture defines the stacked registers as a circular set. Calls and returns move the active window, or stack frame through this set via renaming. Successive calls may move the set into a region of used registers, requiring an operation similar to a traditional register stack 'push'. Successive returns may move the set into a region of registers containing stale data from previously returned calls, requiring a stack 'pop' to restore the caller data for the next return. In order to simplify handling of the stacked registers, the architecture defines a register stack engine or RSE to implement the register stack abstraction, thus providing the software environment an unlimited number of registers by spilling or filling register data to backing store memory. Unnecessary spilling and filling of registers is avoided at procedure call and return interfaces by giving the compiler some control of the register renaming. At a procedure call interface, a new frame of registers is available to the callee, which includes the output registers of the caller. The callee can execute an *alloc* instruction to specify its intent of register usage. If sufficient registers are not available for allocation, the RSE spills registers from earlier frames to memory. This continues until the callee's request can be satisfied. Similarly upon a procedure return, the RSE may have to restore the caller's registers from memory.

Figure 1 shows an example procedure call sequence to illustrate register stacking. In this figure, the numbers to the left of the boxes represent virtual registers (VRn) and the numbers on the right represent the corresponding physical/aliased registers (PRn,Arn). It shows procedure A which has declared virtual registers 32 thru 45 as local or input registers. Virtual registers 46 thru 52 are the output registers of procedure A. Upon calling procedure B, the output registers of procedure A form the initial register frame for procedure B and are now referenced by virtual registers 32 thru 38. By performing an alloc operation, procedure B expands its register frame from virtual register 32 to 50 with virtual registers 48 thru 50 designated as output registers. This process of stacked register renaming is again repeated when procedure B calls procedure C.

In addition, the stacked registers may also participate in register rotation. Register rotation is a feature of the Itanium® architecture to efficiently support modulo scheduling of loops. The modulo scheduling of loops allows compilers to parallelize loop iterations. Register rotation is the software visible register renaming mechanism through which each iteration of a loop is provided its own set of registers.

The aforementioned characteristics of the Itanium® architecture all lead to significant demand on GR stacked register sets. This demand eventually results in increased RSE activity in order to guarantee correctness and provide software the illusion of unlimited registers. As the RSE spill and fill activities ultimately displace work-producing instructions, performance can be increased by minimizing

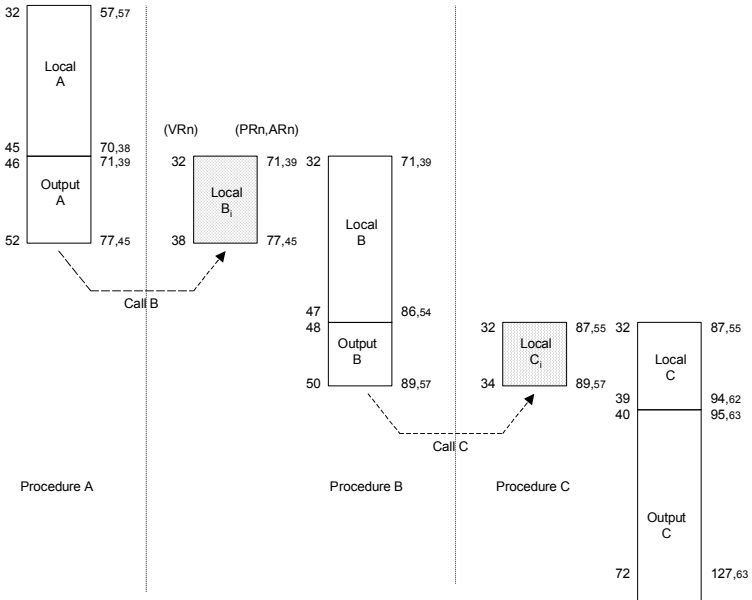

Figure 1: Example call sequence illustrating register renaming and aliasing

RSE activity. This is best accomplished in hardware by increasing the number of physical registers used to implement the stacked register sets.

Given the increasing complexity of contemporary microprocessors, new methods are required in order to limit the implementation expense of data hazard detection. Rather than traditional comparators and scoreboards for hazard detection, a new structure, which integrates both, can be used. Assume that two regids are decoded into 1-hot format; for a 128-entry GR register file a 7-bit binary-encoded regid turns into a 128-bit vector. Hazards can be then detected (in the simplest case) on a per-GR basis with a simple logical AND of the bits from each regid vector corresponding to each GR. The result is a hazard vector containing 0 asserted bits if the two regids were not equal, and a single asserted bit if the regids were equal and thus constitute a data hazard. A bitwise logical OR of the resulting hazard vector is all that is required to signal the hazard. This method would be inefficient if each compare was performed separately, but provides a method for significant hardware reduction. Since a data hazard occurs when ANY consumer regid matches the regid of ANY producer lacking a bypass path, we can combine like terms in detecting RAW hazards. By logically ORing the 1-hot vectors of all REG stage consumers, and logically ORing the 1-hot regid vectors of all older non-bypass producers, we reduce the problem (again, in the simplest case) to a single logical AND of two multi-hot regid vectors, with a bitwise OR required to generate the hazard signal. Such a structure consists of wide logical ORs and 2-input logic ANDs, and is optimal for implementation in dynamic CMOS logic. Similar methods reduce the hardware required for WAW hazard detection.

The entire hazard detection problem is, of course, more complicated than the example given, but is easily performed with multiple of these vector AND functions for different types and cases of hazards. Significant sharing of decode hardware is possible in generating the vectors for the various ANDs needed for complete hazard detection. It should be apparent that most of the 7->128 bit decodes used to generate these regid vectors are the same needed for the various read and write ports to the load scoreboards, thus re-using already required hardware. Regularity of the wiring and of the logic and decode structures further adds to the efficiency of this method of hazard detection. Each ‘row’ of the hazard logic array can be identical, and is associated with one physical register. Note that the various qualifiers can be folded into the decode logic, producing either 1-hot or all-zero vectors, depending upon the qualifier values, adding the last required layer of required complexity to the hazard detection logic.

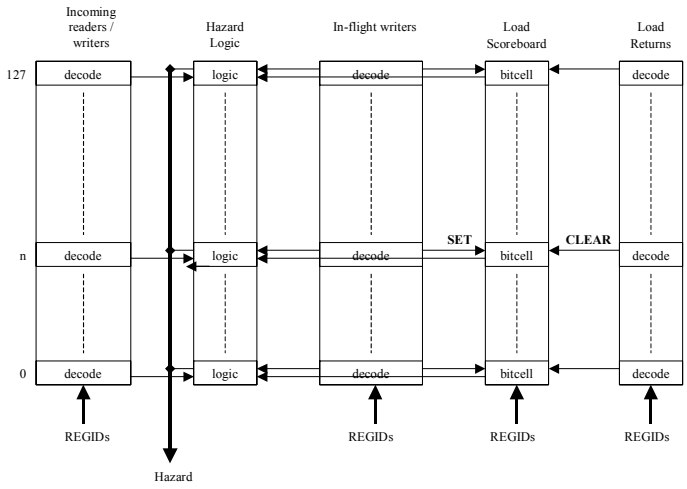

Figure 2: Hazard Logic Array Block Diagram

Figure 2 shows a block diagram of the hazard logic array physical structure. This figure shows columns of logic structures where each row is associated with a single physical register. The figure shows a 128 register high hazard logic array as an example. The far left logic column represents the source operand register identifiers of incoming or consumer instructions. These REGIDs are broadcast up the decoders and then the decoded signals travel across to the hazard logic detection column to combine with similar decode information from in-flight writers and the scoreboard information.

3. Stacked Register Aliasing

The vector-based method of hazard detection greatly simplifies the non-linear dependence of logic complexity upon the number of pipelines and pipeline stages. But it creates a linear dependence of complexity upon the number of physical registers; doubling the register file size roughly doubles the size of the hazard logic array. This dependence upon register file size might preclude increasing the number of stacked physical registers if the size of the hazard logic array grew to an unacceptable size, or if the growth slowed the logic to an unacceptable clock speed.

The size problem of the hazard detection logic may be solved by the introduction of aliasing in the hazard logic. By mapping multiple physical registers to a single row in the hazard logic, we are able to increase the number of physical registers, while maintaining or reducing the size of the hazard logic array. As an example, the 128 GRs can be represented by 64 rows in the hazard logic array. GR[127:64] would be mapped (aliased) by the decoders to the same 64 rows used for GR[63:0] in the hazard logic array. The reduction in complexity is significant, but usages of GR[127] and GR[63] are then indistinguishable. It is easy to see that a false stall can occur because of an apparent hazard caused by a REG-stage consumer of GR[63] and an older producer targeting GR[127].

In order to minimize false stalls we must consider the various types of registers used in an architecture, as well as the register frame size generally in use for those registers. We continue to have separate hazard logic arrays for GRs and FRs because their hazards are generally orthogonal – the pipeline stages and bypass structures differ considerably. In the GRs, though, recall that we have two significantly different register types; static and stacked registers. The static registers may be used at any time by any routine. The stacked GRs, however, are seen in contemporary code to be used in sets smaller than the full virtual set of 96. Even as calls and returns occur we experience a fractional window of the total to generally be referenced by the instructions in-flight at any time. Given these conditions we are best able to utilize aliasing within the stacked registers, while maintaining non-aliased hazard logic rows for the static registers.

The calling sequence shown in Figure 1 illustrates the stacked register aliasing effects. Here, 3-way aliasing i.e.

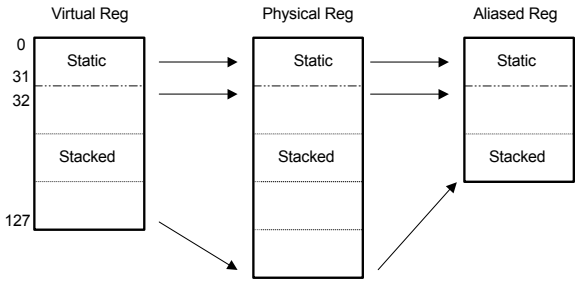

Figure 3: Register ID Mapping

the 96 stacked registers are mapped onto 32 aliased registers, is shown. Procedure B has a register frame size of 19 registers spanning from physical regid of 71 to 89. The corresponding aliased regids are from 39 to 57. Thus, this procedure will not suffer any false hazards due to aliasing. Procedure C is shown with a register frame size of 56 registers. In this case, physical regid 95 and 127 will both map to aliased regid 63. Hence, it is possible for false hazard stalls to be signaled.

If 96 physical registers are used for the 96 virtual GRs, we may represent these in the hazard logic as 48 rows (2:1 aliasing), 32 rows (3:1 aliasing), or even smaller sets, although smaller sets can easily be seen to greatly increase the detection of false data hazards. If 32 rows are used for the stacked registers, a total of 64 rows will result (32 static + 32 stacked). This eliminates 50% of the rows required with no aliasing. If only 48 rows are used, the row total increases to 80, but still represents a 37.5% reduction. These reductions are desirable even if physical register count is not increased. Reducing the number of rows has a significant effect on the power consumption of the circuit, both in the hazard logic and in the decoders. In addition, there is some reduction in circuit delays as well as the obvious reduction in area.

The benefits of aliasing are compelling as the physical register count increases. As we have discussed above, it can be desirable to increase the number of stacked physical registers in an Itanium® processor microarchitecture to reduce the number of register spills and fills. Since hazard detection is performed on physical regids, any increase in the physical registers adds complexity to the hazard logic. Hazard aliasing provides the opportunity to, at worst, minimize the increase in complexity. At best, the hazard logic can be reduced even as the number of physical registers is increased.

For a minimal change, assume that we increase from 96 to 128 the number of physical registers used for the 96 stacked GRs. We may choose the minimum (2:1) aliasing so that 64 stacked and 32 static rows are required. These 96 hazard logic rows still represent a savings of 40% verses the 160 rows otherwise required, and a 25% savings over the original solution with 96 stacked registers. These savings are not available when using explicit binary comparators for hazard detection – a 2:1 aliasing for 160 registers would offer minimal savings as 8-bit comparators simply become 7-bit comparators. Figure 3 shows how virtual registers are mapped onto physical registers and then to aliased register identifiers for hazard detection. The 32 static registers map as is all the way across on the top. The stacked registers shown in Figure 3 assume we use 128 physical registers for the 96 virtual registers and that these are mapped onto 64 hazard logic rows.

Aliasing in vector-based hazard detection is a powerful technique for mitigating the expense of hazard detection as register file sizes increase. It also adds a degree of freedom in selecting the physical size of each register file as power and area budgets can be now be dictated by the actual register file itself. But the implementation of aliasing in hazard detection obviously presents several design trade-offs. Increased aliasing ratios have the potential for performance improvements via power savings and decreased cycle time, but also increase the probabilities of false stalls. Optimizing the design requires careful performance analysis performed within the proper context, both internal and external. Internal factors are primarily related to processor microarchitecture; increased issue width and pipelined depth increases the number of stacked register frames likely to be inflight, and the memory subsystem implementation affects the number of scoreboarded loads (an important detail since these are long-latency loads and

may belong to any number of stack frames). External details are primarily related to application code and compilers, and, to a lesser degree, the operating system.

4. Performance Results

Several experiments were conducted to quantify the expected performance loss due to false hazards introduced by register aliasing. An internal performance simulator which provides a cycle accurate model for a given Itanium® processor microarchitecture was used for these experiments. The workloads were selected from Spec95 suite. Internal HP Itanium Processor Family compilers produced the binaries for these benchmarks.

Each of the benchmarks was first run without any register aliasing to collect a baseline total cycle count. Then, the performance simulator was modified to implement the register aliasing technique. Total cycle counts were collected for 2-way and 3-way aliasing.

The results of the aforementioned experiments are presented in Table 1 and 2. Table 1 shows the relative performance loss due to 2-way and 3-way aliasing for SpecINT95 benchmarks. Table 2 shows the results for SpecFP95 benchmarks.

The integer benchmarks show negligible performance loss for both 2-way and 3-way aliasing. There are a couple of reasons why this might have been expected. First, integer programs do not have as many modulo scheduled loops as technical programs, which would tend toward smaller register frames. Second, typically these programs are scheduled for a single cycle latency cache as is present in the Itanium® 2 processor. If the compiler were to schedule for higher cache latencies, the register live requirements would increase and consequently create larger register frames. The SpecFP95 benchmarks on the other hand do demonstrate some sensitivity to register aliasing. On the whole, SpecFP95 ratios declined 5.34% with 2-way aliasing and 17.48% with 3-way aliasing. One benchmark, 107.mgrid, suffered a significant increase in total cycles. The performance loss on SpecFP95 can be attributed to heavy use of modulo scheduled loops, which tend to require significantly larger register frames for the purposes of register rotation. Furthermore, SpecFP95 benchmarks were scheduled for larger cache latencies, which is on the order of 5-8 cycles.

Based on the above experimental results, one can conclude that the register aliasing technique can be implemented for the integer registers and not for the floating-point registers. Since the floating-point registers are not stacked in the Itanium® architecture, it is not necessary to provide additional physical registers beyond what is architecturally visible. However, on the integer side, the use of this aliasing technique can form the basis of a physically scalable register file implementation. For example, increasing the number of physical stacked integer registers from 96 to 128 could be accomplished with 2-way aliasing.

5. Acknowledgements

The authors would like to specially thank Terry Lyon, Greg Woods and Subramoni Parmeswaran for their assistance in the setup of performance experiments and collection of the performance data highlighted in this paper.

6. References

[1] D. Soltis, R. Bhatia and R. Arnold, “Stacked Register Aliasing in Data Hazard Detection to Reduce Circuit Size,” HP US Patent Application 10016639, Feb 2002.

[2] S. Naffziger, and G. Hammond. The Implementation of the Next Generation 64b Itanium Microprocessor. *Proc of ISSCC*, Feb 2002.

[3] E. Fetzer, and J. Orton. A Fully-Bypassed 6-Issue Integer Datapath and Register File on an Itanium Microprocessor. *Proc of ISSCC*, Feb 2002.

[4] D. Mosberger, and S. Eranian. *IA-64 Linux Kernel Design and Implementation*, Prentice Hall PTR, 2002.

[5] R. Zahir, D. Morris, J. Ross, and D. Hess. OS and Compiler Considerations in the Design of the IA-64 Architecture. *Proc of ASPLOS-IX*, Cambridge, MA, Nov 2000.

[6] Intel Corporation. *Itanium® Architecture Software Developer’s Manual*, http://developer.intel.com/design/itanium/manuals/index.htm.

Table 1: SpecINT95 Results

RegFile Alias	2-way	2-way	3-way	3-way
SpecINT95 Benchmark	**SpecINT95 Ratio** % Change	**SpecINT95 Total Cycles** % Change	**SpecINT95 Ratio** % Change	**SpecINT95 Total Cycles** % Change
099.go	0.00	0.01	-0.02	0.02
124.m88ksim	0.00	0.00	0.00	0.00
126.gcc	0.02	-0.01	-0.02	0.01
129.compress	0.00	0.00	0.00	0.00
130.li	0.00	0.00	0.00	0.00
132.ijpeg	0.00	0.08	0.00	0.00
134.perl	0.00	0.00	0.00	0.00
147.vortex	-0.08	0.03	-0.05	0.05
SpecINT95	-0.02		-0.02	

Table 2: SpecFP95 Results

RegFile Alias	2-way	2-way	3-way	3-way
SpecFP95 Benchmark	**SpecFP95 Ratio** % Change	**SpecFP95 Total Cycles** % Change	**SpecFP95 Ratio** % Change	**SpecFP95 Total Cycles** % Change
101.tomcatv	0.49	-0.49	-14.74	17.29
102.swim	0.00	0.00	-20.85	26.33
103.su2cor	-1.77	1.80	-19.82	24.71
104.hydro2d	-3.81	3.95	-13.55	15.65
107.mgrid	-30.73	44.37	-44.16	79.08
110.applu	-8.42	9.21	-13.11	15.10
125.turbo3d	-0.01	0.02	-8.48	9.26
141.apsi	-2.21	2.26	-15.70	18.62
145.fpppp	-0.32	0.30	-0.59	0.57
146.wave5	-1.66	1.69	-16.00	19.05
SpecFP95	-5.34		-17.48	